Lecture Notes

Lectures 3 & 4

CS389: Machine Learning, Spring 2023

University of Massachusetts Amherst

1 Multilayer Perceptrons (lecture 5)

1.1 Non-linearities

In a previous lecture, we mentioned that the perceptron is unable to solve a dataset that is not linearly separable (e.g XOR). Adding more layers to the network, without any non-linear functions, will be ineffecitve because the model with many layers is still linear and could still be represented as a single layer perceptron.

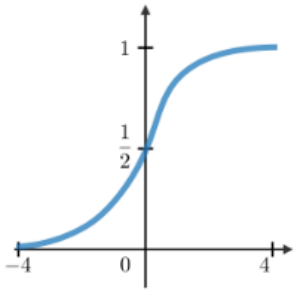
$$W_m(W_{m-1}(W_{m-2}(\cdots W_1 X))) = WX \tag{1.1}$$

Another way we can show this is that if we have 2 distinct linear models $g(x)$ and $f(x)$, then $f(g(x))$ is a linear model that is equivalent to some other linear function $h(x)$, think composite functions.

$$h(x) = f(g(x)) \tag{1.2}$$

To approximate functions that do not follow linearity, we will introduce the concept of non-linear functions. Non-linearity will allow us to create non-linear mappings between the inputs and outputs. The most commonly used non-linear functions are the *sigmoid*, *tanh*, and *ReLU* functions. Other non-linear functions may have different properties and may be useful for specific tasks, but sigmoid, tanh, and ReLU have become popular choices because they are simple, continuous, differentiable, and have other desirable properties for neural networks.

1.1.1 Sigmoid



Non-linearity Function	Derivative
$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma(x) \cdot [1 - \sigma(x)]$

Figure 1: Sigmoid Function

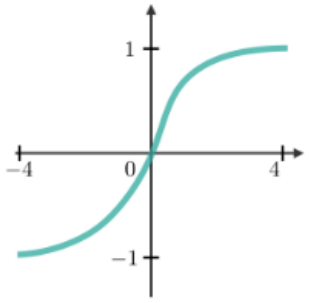
The sigmoid function takes a real-valued number and "squashes" it into range between 0 and 1. In particular, large negative numbers become 0 and large positive numbers become 1. The sigmoid

function has seen frequent use historically since it has a nice interpretation as the firing rate of a neuron: from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1). In practice, the sigmoid non-linearity has recently fallen out of favor and it is rarely ever used. It has two major drawbacks:

- *Vanishing Gradients.* A very undesirable property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. Therefore, if the local gradient is very small, it will effectively "kill" the gradient and almost no signal will flow through the neuron to its weights and recursively to its data. Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large then most neurons would become saturated and the network will barely learn.
- *Outputs not zero-centered.* This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive, then the gradient on the weights w will during backpropagation become either all be positive, or all negative.

1.1.2 Tanh

The tanh function squashes a real-valued number to the range [-1, 1]. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid nonlinearity.



Non-linearity Function	Derivative
$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - \tanh^2(x)$

Figure 2: Tanh Function

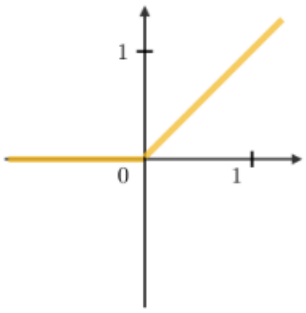
1.1.3 ReLU

The Rectified Linear Unit (ReLU) has become very popular in the last few years. The activation is simply thresholded at zero.

- Unfortunately, ReLU units can be fragile during training and can "die". For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. For example, you may find that as much as 40% of your network can be "dead" (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high. With a proper setting of the learning rate this is less frequently an issue.

2 Backpropagation (lecture 5)

As an example, we can setup a neural network with a single weight in each layer with no bias term like in Figure 4. In each layer, we pass the dot product of the weight and the input through an



Non-linearity Function	Derivative
$\text{ReLU}(x) = \max(0, x)$	$\begin{cases} 0 & x < 0 \\ 1 & \text{otherwise} \end{cases}$

Figure 3: ReLU Function

activation function, A. The output is then passed onto the next layer in the network as an input. For example, the output of the first layer is $A_1(W_1X_1)$ becomes the input for the second layer. For this example we will be using the *sigmoid* activation function.

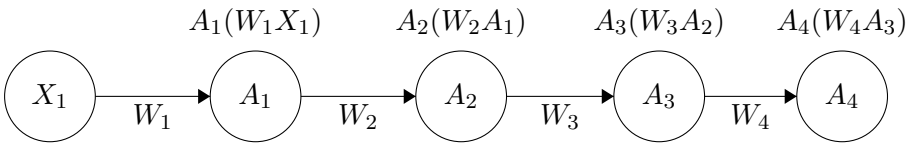


Figure 4: A 4 layer network with a single-weight each layer

Now recall the equation for the gradient descent update step as well as the gradient of loss from the previous lectures:

$$W = W - \alpha \frac{\partial L}{\partial W} \tag{2.1}$$

$$\nabla_W \text{Loss} = \frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_m} \right) \tag{2.2}$$

Previously, we solved this using the chain rule:

$$\frac{\partial L}{\partial w_j} = \frac{\partial \hat{y}}{\partial w_j} \frac{\partial L}{\partial \hat{y}} \tag{2.3}$$

We can extend this chain rule technique and apply it to our network in Figure 4. The idea is to consider all the terms that depend on each other in the network. The reason why we need to do this is because the input of a layer is the output of the previous layer. For example, changing w_2 will affect A_4, A_3, A_2, A_1, and L so we must consider all these variables when we try to find $\frac{\partial L}{\partial w_1}$. We can do this by multiplying all the partial derivatives along the path from loss to w_1:

$$\frac{\partial L}{\partial w_4} = \frac{\partial L}{\partial A_4} \cdot \frac{\partial A_4}{\partial w_4} \tag{2.4}$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial A_4} \cdot \frac{\partial A_4}{\partial A_3} \cdot \frac{\partial A_3}{\partial w_3} \tag{2.5}$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial A_4} \cdot \frac{\partial A_4}{\partial A_3} \cdot \frac{\partial A_3}{\partial A_2} \cdot \frac{\partial A_2}{\partial w_2} \tag{2.6}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial A_4} \cdot \frac{\partial A_4}{\partial A_3} \cdot \frac{\partial A_3}{\partial A_2} \cdot \frac{\partial A_2}{\partial A_1} \cdot \frac{\partial A_1}{\partial w_1} \tag{2.7}$$

Observe that a lot of the partial derivattives are repeated, as colored in blue, this will be important in a bit. From the above, we can see that the partial derivatives generally comes in three forms. Each can be solved likeso (note that A_{m-1} of the first layer is just the input X):

Proof. Solve $\frac{\partial L}{\partial A_m}$ for MSE loss function. Recall that MSE for SGD $= \frac{1}{2}(y - \hat{y})^2$,

$$\begin{aligned} \frac{\partial L}{\partial A_m} &= 2 \cdot \frac{1}{2} \cdot (y - A_m) \cdot (-1) && \text{(chain rule)} \\ &= A_m - y \end{aligned}$$

□

Proof. Solve $\frac{\partial A_{m+1}}{\partial A_m}$ for the sigmoid activation function. Recall that the derivative of the sigmoid function is $\sigma(x) \cdot [1 - \sigma(x)]$ and that $A_{m+1} = \sigma(w_{m+1} \cdot A_m)$.

$$\frac{\partial A_{m+1}}{\partial A_m} = \sigma(A_m) \cdot [1 - \sigma(A_m)] \cdot w_{m+1} \qquad \text{(derivative of sigmoid function)}$$

□

Proof. Solve $\frac{\partial A_m}{\partial w_m}$. Recall that $A_m = \sigma(w_m \cdot A_{m-1})$:

$$\frac{\partial A_m}{\partial w_m} = \sigma(A_{m-1}) \cdot [1 - \sigma(A_{m-1})] \cdot A_{m-1}$$

□

First, notice how the calculation of the partial derivatives of each layer can be computed locally. For example, the partial derivative of $\frac{\partial A_{m+1}}{\partial A_m}$ can be computed in the $(m+1)^{\text{th}}$ layer since the input A_m and weight w_{m+1} is known after the forward pass. Similarly, the partial derivative of $\frac{\partial A_m}{\partial w_m}$ can be calculated in the m^{th} layer since the input to the m^{th} layer is known. Finally, the partial derivatives in $\frac{\partial L}{\partial W} = \{\frac{\partial L}{\partial w_1}, \cdots, \frac{\partial L}{\partial w_m}\}$ has a lot of repeated calculations (as colored in blue). Instead of recomputing the chain at each layer, we can keep track of this chain and continue multiplying onto it. This is also why we want to start at the last layer of the network (e.g $\frac{\partial L}{\partial w_m}$) and go backwards until the first layer (e.g $\frac{\partial L}{\partial w_1}$), hence the name *backpropagation*.

References

[1] Cooper.

[2] Fei-Fei Li, Jiajun Wu, and Ruohan Gao, Stanford CS231n, Spring 2022. https://web.archive.org/web/20230109135558/https://cs231n.github.io/

[3] Graphs of tanh, sigmoid, ReLU. https://studymachinelearning.com/activation-functions-in-neural-network/